Algebra of Behavior Tables[*]

Steven D. Johnson and Alex Tsow

Indiana University Computer Science Department
sjohnson@cs.indiana.edu

Abstract

A design formalization based on behavior tables *was presented at Lfm97. This paper describes ongoing work on a supporting tool, now in development. The goal is to make* design derivation, *the interactive construction of correct implementations, more natural and visually palatable while preserving the benefits of formal manipulation. We review the syntax and semantics of behavior tables, introducing some new syntactic elements. We present a core algebra for architectural refinement, including new notational conventions for expressing such rules.*

KEYWORDS: behavior table, design derivation, formal synthesis.

1. Introduction

Behavior table notation emerged out of case studies in formal *design derivation* between 1985 and 1995. The *DDD transformation system* [7] is based on functional algebra. Behavioral expressions at the level of *algorithmic state machines* [1] are represented by recursive systems of function definitions, and architecture oriented implementations are represented by recursive systems of stream expressions. In DDD, these representations are manipulated as transformations on Scheme programs, so the expressions are also executable.

The primary goal in our early case studies was to interactively impose hardware architectures on algorithmic specifications. As these studies became larger, a practice emerged of printing DDD expressions in a tabular form, reminiscent of register transfer tables. The tables helped design teams visualize their architectural goals so they could strategize about how to accomplish them in the DDD algebra.

We began to contemplate using the tables more directly as formal objects, retargetting the DDD algebra to operate on tabular representations. We believe the tables are more perspicuous to practicing professionals who, it has been claimed, are put off by the notation used in formal reasoning systems.

The rising visibility of tabular specification languages such as *Tablewise* [3], *SCR** [2], and *and-or* transitions in *RSML* [8], helped convince us to look at behavior tables more seriously as a formalism rather than merely as a visual aid. Subsequently, we have undertaken to develop a tool for interactive design derivation using them.

In this paper, we develop a core algebra for architectural manipulation. In the main, this algebra correlates to the "structural" algebra of *sequential systems*, presented in [5]. Although the main purpose is to lay the groundwork for tool implementation, one ancillary contribution of this paper is its notational conventions for stating the rules of the algebra, which use *table schemes* to simplify quantification.

The conclusion lists additional topics and issues entailed in the implementation effort. We extend the term-level syntax presented at Lfm97 [6] to include provisions for *bounded indirection*, additional algebra for a simple kind of *data refinement*, and possible extensions for *verification*.

[*] This research is supported, in part, by the National Science Foundation under Grant MIP9610358.

2. Terms

Behavior tables are arrays of *terms* in a ground vocabulary of constants and operations. We very briefly review the terminology of first order structures then introduce the extensions that are assumed in behavior tables.

A *first order structure* describes a family of value sets, $A_1, \ldots, A_n$, together with a collection of total functions, $f_1, \ldots, f_m$, on these sets. With each set A_i is associated a type symbol, τ_i,. There are *constant* and *operator* symbols representing the functions f_i, and a distinct set of *variable* symbols. The notation $v\colon \tau_i$ asserts that the variable v ranges over values in A_i. The *signature* of an operator specifies its domain and range, which in general are nested products. The formula $f\colon(\tau_1,(\tau_2,\tau_3)) \to (\tau_4,\tau_5,\tau_6)$ asserts that the operation f maps the product $A_1 \times (A_2 \times A_3)$ to the product $A_4 \times A_5 \times A_6$. We shall allow for *multioutput* operations, as suggested here, whose output signatures are n-tuples.

A *term* is a variable, constant, or application, $f(T_1, \ldots, T_n)$, of an operation f to the terms T_i according to the f's signature.

A structure becomes an equational algebra when it is provided with a set E of equational *identities* among terms (over a distinguished set of *logical variables*). E induces an equivalence relation; and we write $\models_E s \equiv t$ to express the fact that s and t are provably equivalent under E.

Certain additional features are assumed of all structures used in behavior tables and are thus absorbed at the metalinguistic level.

- A sort Bool is assumed with constants true and false and the identities of *boolean algebra.* Operations with range Bool are called *tests.*

- A *don't care* constant is designated by '♮'.

- Finite product (tupling) and projection operations of each type are assumed Projections are denoted by sans-serif adjectives, $\mathsf{1st}$, $\mathsf{2nd}$, $\mathsf{3rd}$, $\mathsf{4th}$, $\mathsf{5th}$ ..., $i\mathsf{th}$, ... An n-tuple is expressed as a parenthesized series of n terms, $(T_1, \ldots, T_n)$. Projections applied to n-tuples can be simplified at the syntactic level; for instance,
$$\models \mathsf{2nd}(T_1, T_2, T_2) \equiv T_2$$

- It is assumed that arbitrary finite sets of *tokens* can be represented (e.g. by n-tuples over Bool). We shall extend this idea to what Hoover calls a *finite logic* [3], with which we associate a specific selection operation, written
$$\begin{array}{l} \mathsf{case}\ s\ \mathsf{of} \\ \quad \begin{array}{rcl} \mathsf{a}_1 & : & t_1 \\ & \vdots & \\ \mathsf{a}_k & : & t_1 \end{array} \end{array}$$

The usual treatment of terms is extended for explicit multioutput operations. The definition of substitution on terms is adapted for multioutput operations by allowing nested lists of variables to serve as substitution patterns. Such a list is called an *identifier.*

Definition 1 *An* identifier *is either a variable or a nested list,* $(X_1, \ldots, X_n)$, *of* distinct *identifiers, meaning that they share no common variables.*

Definition 2 *The formula* $T[R/X]$ *denotes a* substitution *of the term* R *for the identifier* X *in the term* T. *The formula* $T[R_1/X_1, \ldots, R_n/X_n]$ *denotes the simultaneous and respective substitutions of terms* R_i *for identifiers* X_i, $i \in \{1\,..\,n\}$. *Substitution is defined by induction on the language of terms. In the base cases, constants are unchanged and for a variable symbol* u,
$$u[R/X] = \begin{cases} R & \text{if } X = u \\ u & \text{if } X \neq u \end{cases}$$
For applications and n*-tuples,*
$$f(T_1, \ldots, T_n)[R/X] = f(T_1[R/X], \ldots, T_n[R/X])$$

For nested identifiers, a simultaneous substitution is done on the constituents:
$$T[R/(X_1, \ldots, X_n)] = T[\mathit{1st}(R)/X_1, \ldots, n\mathit{th}(R)/X_n]$$
In the last case, substitution of an n-tuple for an n-element identifier simplifies to
$$\begin{array}{c} T[(R_1, \ldots, R_n)/(X_1, \ldots, X_n)] \\ = T[R_1/X_1, \ldots, R_n/X_n] \end{array}$$

3. Syntax of behavior tables

Behavior tables are closed expressions whose terms contain variables from three disjoint sets: I (inputs), S (sequential signals, or data state), and C (combinational signals). Fix these sets for the remainder of this section. We will write ISC for $I \cup S \cup C$ and SC for $S \cup C$. We use the term "register" for an element of S, but this is a euphemism that should be interpreted very abstractly. There is no assumption that these variables denote finite values, nor are tables intended only for register-transfer specification. The form of a behavior table is:

Name: *Inputs* → *Outputs*	
Conditions	*Registers and Signals*
⋮	⋮
Guard	*Computation Step*
⋮	⋮

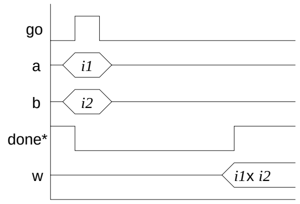
Inputs is a list of input variables and *Outputs* is a set of terms over ISC, but without loss of generality, assume $O \subseteq SC$. *Conditions* is a set P of predicates over ISC, that is, finitely typed terms ranging over finite types, such as truth values, token sets, etc.

The notion of term evaluation used here is standard. The value of a term, t, is written $\sigma[\![t]\!]$, where σ is an *assignment* or association of values to variables.

Definition 3 *A* guard *is a set of constants indexed by a condition set P: $g = \{c_p\}_{p \in P}$. A* decision table $\mathbf{D} = [P, G]$, *consists of a condition set and a an associated list of guards. We say g* holds *for an assignment σ to ISC when, for each $p \in P$, either $c_p = \natural$ or $\sigma[\![p]\!] = c_p$.*

Following [3], we say a decision table is *functional* when G describes a proper partitioning of the possible assignments to ISC. In other words, the guards are "consistent" and "complete".

Definition 4 *A* computation step *or* action *is a set of terms, one for each register and signal: $a = \{t_v\}_{v \in SC}$. An* action table *is a set of actions typically indexed by the guards of a corresponding decision table.*



MULT:(go, a, b) → (done*, w)						
go	P	(even? u)	u	v	w	done*
1	♮	♮	a	b	0	$P \wedge \neg$go
0	1	♮	♮	♮	w	$P \wedge \neg$go
0	0	1	u÷2	v×2	w	$P \wedge \neg$go
0	0	0	u÷2	v×2	w+v	$P \wedge \neg$go

where $P \equiv$ (zero? u) $\vee$ (zero? v)

Figure 1: Example of a behavior table

Definition 5 *A* behavior table *for $I \to O$ consists of a decision table, $\mathbf{D}$, with guards $G = \{g_1, \ldots g_n\}$, and an action table indexed by G, $\mathbf{A} = \{t_{v,k} \mid v \in SC$ and $g_k \in G\}$.*

Figure 1 shows a shift-and-add multiplier, expressed as a behavior table. The timing diagram is provided to explain the interface, with multiplication performed within a full handshake.

4. Synchronous semantics

A behavior table $[\mathbf{D}, \mathbf{A}]$ for $O \subseteq SC$ denotes a relation between infinite input and output sequences. We call these sequences *streams* because in prior work we obtain a semantics by interpreting a table as a (co)recursive system of stream-defining equations [7]. More directly, suppose we are given a set of initial values for the registers, $\{x_s\}_{s \in S}$ and a stream for each input variable in I. Construct a sequence of assignments, $\langle \sigma_0, \sigma_1 \ldots \rangle$ for ISC as follows:

(a) $\sigma_n(i)$ is given for all $i \in I$ and all n.

(b) For each $s \in S$, $\sigma_0(s) = x_s$.

(c) $\sigma_{n+1}(s) = \sigma_n[\![t_{s,k}]\!]$ if guard g_k holds for σ_n.

(d) For each $c \in C$, $\sigma_n(c) = \sigma_n[\![t_{c,k}]\!]$ if guard g_k holds for σ_n.

The stream associated with each $o \in O$ is $\langle \sigma_0(o), \sigma_1(o), \ldots \rangle$. This semantic relation is well defined if there are no circular dependencies among the combinational actions $\{t_{c,k} \mid c \in C,\ g_k \in G\}$. The relation is a function (i.e. deterministic) if decision table $\mathbf{D}$ is functional. We shall restrict our attention to behavior tables that are *well formed* in these respects. In essence, well formedness reflects the usual properties required of synchronous finite state machines.

To achieve well formedness, we constrain behavior tables in two ways. First, we prohibit "combinational feedback" in the actions. Given row k in the action table $\{t_{v,k} \mid v \in SC\}$, there is a natural dependence graph with vertices corresponding to the signal names and edges given by the relation: $a \to b$ iff a is a subterm of $t_{b,k}$. Checking for combinational cycles is a straightforward depth-first search.

Even if the actions themselves do not contain combinational loops, the decision table can still induce race conditions or metastable behavior. Consider the following table fragment where r and c are registered and combinational boolean signals:

$B : I \to O$					
r	c*	$\cdots$	r	c*	$\cdots$
0	0		0	1	
0	1		0	0	
1	0		1	1	
1	1		1	1	

Intuitively, if the system makes a transition into a state where $\sigma_n(r) = 0$, then combinational signal a will oscillate. Our semantics is not well defined in this case: if $c_r = 0$ and $c_c = 0$ in some guard $g_k = \{c_p\}_{p \in P}$ at timeslice n, then $\sigma_n(c) = 1$ by (d). Since g_k no longer holds at σ_n, some other guard $g_j = \{d_p\}_{p \in P}$ in which $d_r = 0$ and $d_c = 1$ hold changes $\sigma_n(c)$ back to 0.

The race condition occurs in our example when $\sigma_n(r) = 1$ and $\sigma_n(c) = 0$. Although one could argue that σ_n is well defined, we shall prohibit this mode of expression anyway, as it reflects a kind of transition race.

To eliminate these scenarios, we constrain the predicates of the decision table to use only registered variables and input signals. This way, no action can directly change the guard g_k since the values of registered signals persist for the duration of the present action (c).

In addition, we shall require a functional set of guards, as noted earlier. This results in deterministic and total behavior, for which the algebra presented here is intended.

We think of behavior tables as denoting persistent, communicating processes, rather than subprocedures. In other words, behavior tables cannot themselves be entries in other behavior tables, but instead are composed by interconnecting their I/O ports. Composition is specified by giving a connection map that is faithful to each component's arity. In our function-oriented modeling methodology, such compositions are expressed as recursive systems of equations,

$$\lambda(U_1, \ldots, U_n).(V_1, \ldots, V_m) \textit{ where}$$

$$\begin{array}{rcl} (X_{11}, \ldots, X_{1q_1}) & = & \mathcal{T}_1(W_{11}, \ldots, W_{1\ell_1}) \\ & \vdots & \\ (X_{p1}, \ldots, X_{pq_p}) & = & \mathcal{T}_p(W_{p1}, \ldots, W_{p\ell_p}) \end{array}$$

in which the defined variables X_{ij} are all distinct, each $\mathcal{T}_k$ is the name of a behavior table or other composition, and the outputs V_k and internal connections W_{ij} are all simple variables coming from the set $\{U_i\} \cup \{X_{jk}\}$.

Valid systems must preserve I/O directionality, excluding both combinational cycles and output conflicts. Checking validity has two stages and is again a graph problem:

1. For each behavior table let its inputs and outputs be vertices, and let $i \to o$ when output signal o combinationally depends on input signal i.

2. Add the following edges to the disjoint union of the behavior table I/O graphs: $o \to i$ when $\mathcal{T}_j(\ldots, o, \ldots)$ is the right hand side of an equation where $\mathcal{T}_j$'s I/O signature is $\mathcal{T}_j : (\ldots, i, \ldots) \to O$.

A legitimate connection network exists when this graph has no cycles.

Provided they are well formed, deterministic systems are readily animated in modeling languages that allow recursive stream networks to be expressed [4]. As long as each register has an initial value, the streams are constructed head-first as a fixed-point computation. Translation to both cycle-based and event-based simulation languages is also relatively straightforward, as long as the systems are expressed over the concrete data types these tools recognize.

A synchronous semantics is simple and suited to the clocked implementation models most high-level synthesizers use. In fact, behavior tables will acquire a range of semantics, depending on their applications, just as HDLs and programming languages do. Even with a variety of interpretations, their inherent structure helps reduce the mathematical bookkeeping that often obscures semantic definitions.

5. Behavior Table Algebra

The collection of transformation rules presented in this section applies to architectural refinement. This set is not claimed to be complete nor is minimal in any mathematical sense. At this stage, our principal object is to build a set of rules that is robust enough to serve as a core rule set for tool implementation. mathematical efficiency is a secondary concern, for the moment.

5.1. Notational conventions

Defining these rules has led to some stimulating notational issues. In attempting to present the rules in a clear way, we have been led to consider some novel conventions for expressing features, particularly for quantification. For reasons of both typography and clarity, we want to reduce use of ellipses, columns, and subscripts to describe a table as, for example,

	$b\colon (I_1,\ldots,I_k) \to (O_1,\ldots,O_\ell)$					
	P_1	$\cdots$	P_m	S_1	$\cdots$	S_p
1	g_{11}	$\cdots$	g_{1m}	t_{11}	$\cdots$	t_{1p}
	$\vdots$	$\ddots$		$\vdots$	$\ddots$	
n	g_{n1}		g_{nm}	t_{n1}		t_{np}

Our *table scheme* notation uses the table itself as a quantifier, and uses set elements as indexes rather than number ranges. Uppercase italic variables denote sets; and differently named sets are always assumed to be finite and disjoint. Lowercase *italic* variables denote indices ranging over sets of the same name. The form

$$\begin{array}{cc} & S \\ R & \boxed{x_{rs}} \end{array}$$

represents a two-dimensional array (table) of items, $\{x_{rs} \mid r \in R \text{ and } s \in S\}$. A san seriff[1] identifier denotes a fixed (throughout the scope of the rule) element from the set of the same name. Thus, the form

$$\begin{array}{cc} & \mathsf{s} \\ R & \boxed{x_{r\mathsf{s}}} \end{array}$$

represents a column, $\{x_{r\mathsf{s}} \mid r \in R\}$, and similarly for rows.

Under these conventions, the table scheme from Section 3 looks like

	b: I → *O*	
	P	*S*
1 ⋮ *N*	$g_{n,p}$	$t_{n,s}$

The use of ellipses $1 \cdots N$ on the left is not necessary, but serves as an reminder that the rows are typically numbered. That is, we usually take the set N to be the first "N" numbers.

5.2. The rules

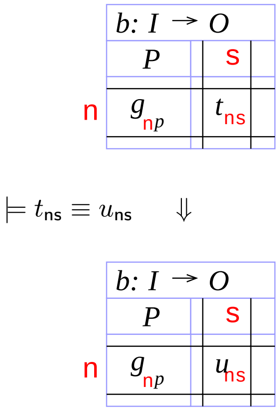
Some structural rules subsumed by the semantics, must be implemented in the tool. For example, interchanging rows and columns is allowed since indices

[1] Where possible, we display these identifiers in red.

range over sets, not sequences. The underlying semantics remain well defined because the order of equations in a system is irrelevant. Similarly, *renaming* variables is allowed under the usual rules of α substitution[2].

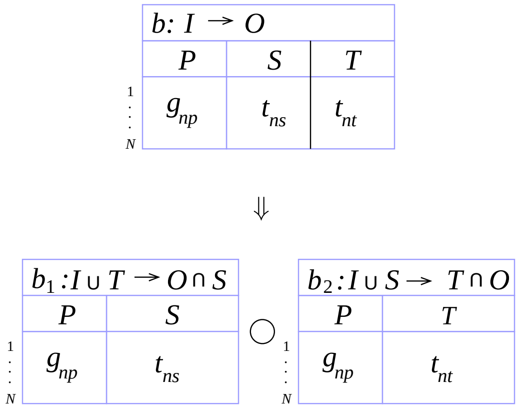
The rules fall into three groups, the first involving both the decision and action table parts, the second being operations on the action table part, and the third being operations that affect the decision table part.

Replacement



One term can be replaced by another term that is (proven to be) equivalent in the underlying structure (or theory). Recall that $\models t \equiv u$ is a provable equivalence in the underlying structure. In practice, establishing equivalence would be done with a rewriting tool or proof assistant.

Decomposition



Decomposition splits one table into two, both inheriting the same decision table. The *compose* operator connects the two tables to maintain the original dependence among the signals. Interpreting the tables as functions on streams—and reading '∪ and '∩ as list operations—$\mathcal{B}_1 \circ \mathcal{B}_2$ yields the system

$$\begin{aligned}&\mathcal{B}(I) \overset{\text{def}}{=} O \ \textit{where}\\ &\quad (O \cap S) \;=\; \mathcal{B}_1(I \cup T)\\ &\quad (O \cap T) \;=\; \mathcal{B}_2(I \cup S)\end{aligned}$$

It is a background job of the table editor to maintain the connection hierarchy as a byproduct of decomposition. An upward *composition* transformation (⇑), if formulated, would require conditions to exclude name clashes and preserve well formedness. In using tables for design derivation, one would typically decompose tables rather than compose them.

This is by no means all there is to say about composition. This strong (in the sense of not being very general) form of the Decomposition rule is essentially a partitioning rule, allowing one to to impose hierarchy on designs.

[2] Actually, behavior tables do not have free variables, so α conversion is even simpler.

Conversion

	$b\colon I \rightharpoonup O$		
	p	q	s
J	$g_{j\mathsf{p}}$	$v_{j\mathsf{q}}$	$t_{j\mathsf{s}}$

$$\Updownarrow \qquad \begin{array}{l} \forall j, j' \in J\colon g_{j\mathsf{p}} \equiv g_{j'\mathsf{p}} \\ \bigcup_{j\in J} v_{j\mathsf{q}} = \mathrm{dom}(\mathsf{q}) \end{array}$$

$b\colon I \rightharpoonup O$		
p	q	s
g_{p}	$\natural$	$\mathsf{case}\ \mathsf{q}\ \{\, v_{j\mathsf{q}} : t_{j\mathsf{s}} \}_J$

This rule, allowing function to be moved between the decision and action parts of a table, provides the means to change the boundary between control and architecture. The side conditions say that, within the range indicated by J, the guards outside column q agree, and the guards within column q are exhaustive.

Action collation

	$b\colon I \rightharpoonup O$				
	P		s	r	
$1 \vdots N$	g_{np}		$t_{n\mathsf{s}}$	$t_{n\mathsf{r}}$	

$$\Downarrow \qquad \begin{array}{l} (\mathit{defined}) \\ (\mathit{compatible}) \\ (\mathit{well\ formed}) \end{array}$$

	$b\colon I \rightharpoonup O$				
	P		s	r	
$1 \vdots N$	g_{np}		$t_{n\mathsf{s}} \diamond t_{n\mathsf{r}}$	s	

The idea behind collation is that two, or several, compatible signals can be merged into one by instantiating don't-cares. The '$\diamond$' operator denotes term-level instantiation,

$$t \diamond t' = \begin{cases} t & \text{if } t' = \natural \\ t' & \text{if } t = \natural \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Compatible means that both variables must be combinational or both must be sequential. If both signals are combinational, an audit is required to assure that the resulting system remains well formed, that is, that instantiation does not introduce feedback.

Action identification

	$b\colon I \rightharpoonup O$	
	P	S
$1 \vdots N$	g_{np}	t_{ns}

$$\mathit{combinational} \quad \Updownarrow$$

	$b\colon I \rightharpoonup O$		
	P	S	y
$1 \vdots N$	g_{np}	$t_{ns}\ [\,\mathsf{y} \,/\, r_{n\mathsf{y}}]$	$r_{n\mathsf{y}}$

In terms of systems, this is the recursion rule, stating that y is equal, in a logical sense, to its defining equation, and hence that one can be replaced for the other. In fact, this rule can be applied on a row-by-row basis, but we give the full-column version to reflect the more typical case when a common subterm is being identified. If y were a sequential variable, it would acquire the value $r_{n\mathsf{y}}$ in the next step and so the replacement is invalid.

Action introduction

	b: I → O	
	P	S
1 ⋮ N	g_{np}	t_{ns}

y *fresh*
well formed ⇕ y *unused*

	b: I → O		
	P	S	y
1 ⋮ N	g_{np}	t_{ns}	$r_{n\mathsf{y}}$

A new action column can be added (⇓) as long as the signal name is not redundant and, in the case of combinational signals, the action terms do not refer to the signal being introduced.

Action grouping

	b: I → O	
	P	s
1 ⋮ N	g_{np}	$t_{n\mathsf{s}}$

(*both comb. or both seq.*) ⇕

	b: I → O		
	P	"1(s)"	"2(s)"
1 ⋮ N	g_{np}	$1(t_{n\mathsf{s}})$	$2(t_{n\mathsf{s}})$

Columns can be grouped and ungrouped as long as the resulting columns are purely sequential or purely combinational. Thus, one canonical form for action tables has just two columns. Recall that signals names are nested identifiers; the notation ‘"1(s)"’ means that ungrouping transformations require *explicit* tuples in the header fields, and destructure them in the obvious way. For instance, if s ≡ (a, b) the ungrouped columns will be headed with a and b.

Action table entries need not be explicit tuples, although they can be, because 1, 2, etc. are legitimate operators.

Decision grouping

	b: I → O	
	p	S
1 ⋮ N	$(d_{n\mathsf{p}}, e_{n\mathsf{p}})$	t_{ns}

compatible ⇕

	b: I → O		
	1(p)	2(p)	S
1 ⋮ N	$d_{n\mathsf{p}}$	$e_{n\mathsf{p}}$	t_{ns}

As with action tables, decision table columns can be grouped into tuples. In contrast, the entries are values and the headers are terms, so explicit use of detupling projectors is allowed in both.

Decision introduction

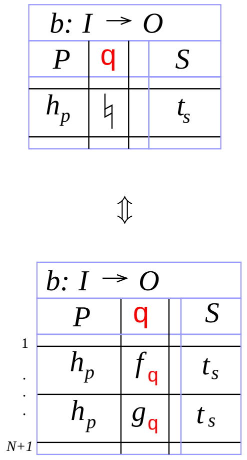
	b: I → O	
	P	S
1 ⋮ N	g_{np}	t_{ns}

Q finite ⇕

	b: I → O		
	P	Q	S
1 ⋮ N	g_{np}	$\natural_{nq}$	t_{ns}

One can introduce a new test with don't-care criteria. The underlying intent of this rule is its use in ad hoc table constructions. A possible well formedness restriction on this rule is that the resulting table be safe from race conditions. Such a restriction can, in principle, be applied when decisions are instantiated (see just below), yielding a more general algebra.

Decision instantiation



Having introduced a new test to a behavior table, instantiation is used to do case splitting. In the simplest case, suppose that a ♮ appears in a decision table entry. Then this rule provides for expanding that row into enough duplicates to account for all the possible values of the test. In the upward direction, the rule gives us a way to combine rows whose actions are identitical. The notation $f_{\mathsf{q}} \cup g_{\mathsf{q}}$ anticipates allowing for decision table entries to be sets of values, as is seen in requirements specification languages.

I/O restriction

Input and output signals may be added to behavior tables without concern so long as the inputs and outputs of the encapsulating system remain the same. Such additions cannot introduce combinational feedback until they are used, and the decision/action introduction rules check for well formedness.

Conversely, an unused I/O signal qualifies for removal. We can remove input i to a behavior table if no action or predicate contains i as a subterm. A behavior table output may be removed when is unused in the surrounding interconnect expression.

6. Other aspects

This paper has developed a core algebra of behavior table manipulation for architectural refinement. In practice, the product of such manipulation is a decomposition of the specification into subsystems for synthesis into hardware or compilation into embedded software components. This section briefly describes a number of other immediate issues and aspects entailed in the development of a design tool.

Figure 2 shows a derivation decomposing a behavior table into two components, one allocating two arithmetic operations to a single device. This is an example of a *system factorization*, a fundamental transformation in the DDD algebra [5], and the instance in the figure comes from an illustration in Johnson's *Lfm97* presentation of behavior tables. The example shows that the algebraic rules presented in this paper are much more finely grained than the transformations that typically would be used in an interactive setting, but would instead serve as a core set of rules from which larger-scale ones are composed.

6.1. Stream semantics

Given a behavior table, one can construct an equivalent sequential system by repeated applications of the Decomposition and Conversion rules. Use Decomposition to separate every column of the action table, then Conversion to reduce each of the resulting tables to a single row. The resulting nested system description can be flattened and simplified. Alternatively, Decomposition can be generalized to simultaneously split tables into several components. To complete the transformation, we must make initialization of the sequential signals explicit. The resulting system is

$$\mathcal{B}(I) \overset{\text{def}}{=} O \textit{ where } \left\{ \begin{array}{lcl} X_{\mathsf{s}} &=& x_{\mathsf{s}} \,!\, \mathsf{select}(\textit{tests}, \textit{alternatives}) \\ Y_{\mathsf{c}} &=& \mathsf{select}(\textit{tests}, \textit{alternatives}) \end{array} \right\}_{\mathsf{s}\in S,\, \mathsf{c}\in C}$$

where the expression $v \,!\, S$ denotes an initialized stream [5]. In DDD, this construction is reversed. An initial behavior table is built from a system of stream equations, each with a common selection combination [7].

6.2. Bounded indirection

We have found an extension to the term-level syntax called *indirection* [11] which is highly useful for hardware applications and appears to be equally useful in incremental specification development. If v is a signal name, the term ${}^\vee v$ stands for a "reference" to signal v; concretely, it is actually a token which can later be used to select v. The term ${}_\wedge w$ denotes that selection. As an illustration, consider the table:

	$I \to O$					
	P	s	t	u	v	S
1		${}^\vee$t	f_1	h_1	♮	
2		${}^\vee$u	f_2	h_2	♮	
3		♮	f_3	h_3	${}_\wedge$s	

In essence, the term ${}_\wedge$s in the third row stands for the term:

$$\begin{array}{l}\mathsf{case\ s}\\ \quad {}^\vee\mathsf{t} : : \mathsf{t}\\ \quad {}^\vee\mathsf{u} : : \mathsf{u}\end{array}$$

Uses of indirection include the description of bidirectional buses, other forms of implied selection, and control branching. Of course, such use also necessitates consistency audits over the whole table; for instance, to verify that selected signals are compatible and uniformly typed.

6.3. Data refinement

Another important set of rules for *data refinement*, will be presented in a future paper. Data refinement involves the translation between levels of data abstraction. In our approach, the foundation for data refinement lies in algebraic specification and equational logic. Consequently, the initial connection to the architectural rules presented here will lie in a more general version of the Replacement rule.

However, replacement is only adequate for the straightforward, combinational expansion of simple representations; it does not address implementations that involve sequential behavior. Our research on *sequential decomposition* [10] has not yet been reflected in behavior tables.

6.4. Verification

We are also interested in integrating the derivational formalism with *property verification.* One way to approach this is to augment behavior tables with *assertions* in a suitable temporal logic. Since we are primarily interested in higher levels of specification, "model checking" [12] these assertions would likely require interaction. Considered as an algorithmic state machine, the table would provide contextual information making the proof process more agreeable.

6.5. Animation

Finally, animation, particularly symbolic execution, would be an important feature of any practical behavior table tool. Consequently, we want to integrate our tool with proof assistants—particularly term rewriters—not only to support replacement rules, verification and type inference, but to provide interactive simplification of terms in the fashion of Moore's *symbolic spread sheets* [9].

References

[1] Christopher R. Clare. *Designing Logic Systems Using State Machines.* McGraw-Hill, 1973.

[2] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: a toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.

[3] D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. Contractor Report 4723, National Aeronautics and Space Administration Langley Research Center (NASA/LRC), Hampton VA 23681-0001, November 1994.

[4] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, Cambridge, 1984.

[5] Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989.

[6] Steven D. Johnson. A tabular language for system design. In C. Michael Holloway and Kelly J. Hayhurst, editors, *Lfm97: Fourth NASA Langley Formal Methods Workshop*, September 1997. NASA Conference Publication 3356, in press.

[7] Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *I-FIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).

[8] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[9] J Strother Moore. Symbolic simulation: an ACL2 approach. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD'98)*, pages 334–350. Springer LNCS 1522, 1998.

[10] Kamlesh Rath, Venkatesh Choppella, and Steven D. Johnson. Decomposition of sequential behavior using interface specification and complementation. *VLSI Design Journal*, 3(3-4):347–358, 1995.

[11] M. Esen Tuna, Kamlesh Rath, and Steven D. Johnson. Specification and synthesis of bounded indirection. In *Proceedings of the Fifth Great Lakes Symposium on VLSI (GLSVLSI95)*, pages 86–89. IEEE, March 1995.

[12] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Ait Mohamed. Model checking for a first-order temporal logic using multiway decision graphs. In *CAV'98*. Springer, 1998.

FIB: (go, in) → (done*, v)						
now	u=0	now	done*	u	v	w
1	♮	done*	¬go	in	0	1
0	1	"	u=0	♮	v	♮
0	0	2	false	u-1	v	w
2	♮	done*	u=0	u	w	v+w

action introduction ⇒

FIB: (go, in) → (done*, v)										
now	u=0	now	done*	u	v	w	x*	y*	z*	ao*
1	♮	done*	¬go	in	0	1	♮	♮	♮	P
0	1	"	u=0	♮	v	♮	♮	♮	♮	P
0	0	2	false	ao	v	w	sub	u	1	P
2	♮	done*	u=0	u	w	ao	add	v	w	P

where $P =$(case x* y*+z* y*-z*)

decomposition ⇒

FIB: (go, in, ao) → (done*, v, now, u, w, x*, y*, z*)									
now	u=0	now	done*	u	v	w	x*	y*	z*
1	♮	done*	¬go	in	0	1	♮	♮	♮
0	1	"	u=0	♮	v	♮	♮	♮	♮
0	0	2	false	ao	v	w	sub	u	1
2	♮	done*	u=0	u	w	ao	add	v	w

output restriction ⇒

FIB: (go, in, ao) → (done*, v, x*, y*, z*)									
now	u=0	now	done*	u	v	w	x*	y*	z*
1	♮	done*	¬go	in	0	1	♮	♮	♮
0	1	"	u=0	♮	v	♮	♮	♮	♮
0	0	2	false	ao	v	w	sub	u	1
2	♮	done*	u=0	u	w	ao	add	v	w

ALU: (go, done*, v, u, now, w, x*, y*, z*) → (ao*)		
now	u=0	ao*
1	♮	(case x y+z y-z)
0	1	(case x y+z y-z)
0	0	(case x y+z y-z)
2	♮	(case x y+z y-z)

input restriction ⇒

ALU: (u, now, x, y, z) → (ao*)		
now	u=0	ao*
1	♮	(case x y+z y-z)
0	1	(case x y+z y-z)
0	0	(case x y+z y-z)
2	♮	(case x y+z y-z)

decision generalization ⇒

ALU: (u, now, x, y, z) → (ao*)		
now	u=0	ao*
1	♮	(case x y+z y-z)
0	♮	(case x y+z y-z)
2	♮	(case x y+z y-z)

decision generalization ⇒

ALU: (u, now, x, y, z) → (ao*)		
now	u=0	ao*
♮	♮	(case x y+z y-z)

decision introduction ⇒

ALU: (u, now, x, y, z) → (ao*)			
x	now	u=0	ao*
♮	♮	♮	(case x y+z y-z)

conversion ⇒

ALU: (u, now, x, y, z) → (ao*)			
x	now	u=0	ao*
add	♮	♮	y+z
sub	♮	♮	y-z

decision elimination ⇒

ALU: (u, now, x, y, z) → (ao*)	
x	ao*
add	y+z
sub	y-z

input restriction ⇒

ALU: (x, y, z) → (ao*)	
x	ao*
add	y+z
sub	y-z

Figure 2: A factorization from [6]